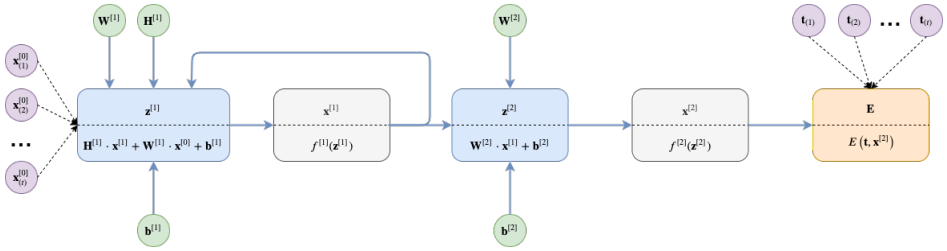
Deep Learning (IST, 2021-22)

Practical 9: Recurrent Neural Networks

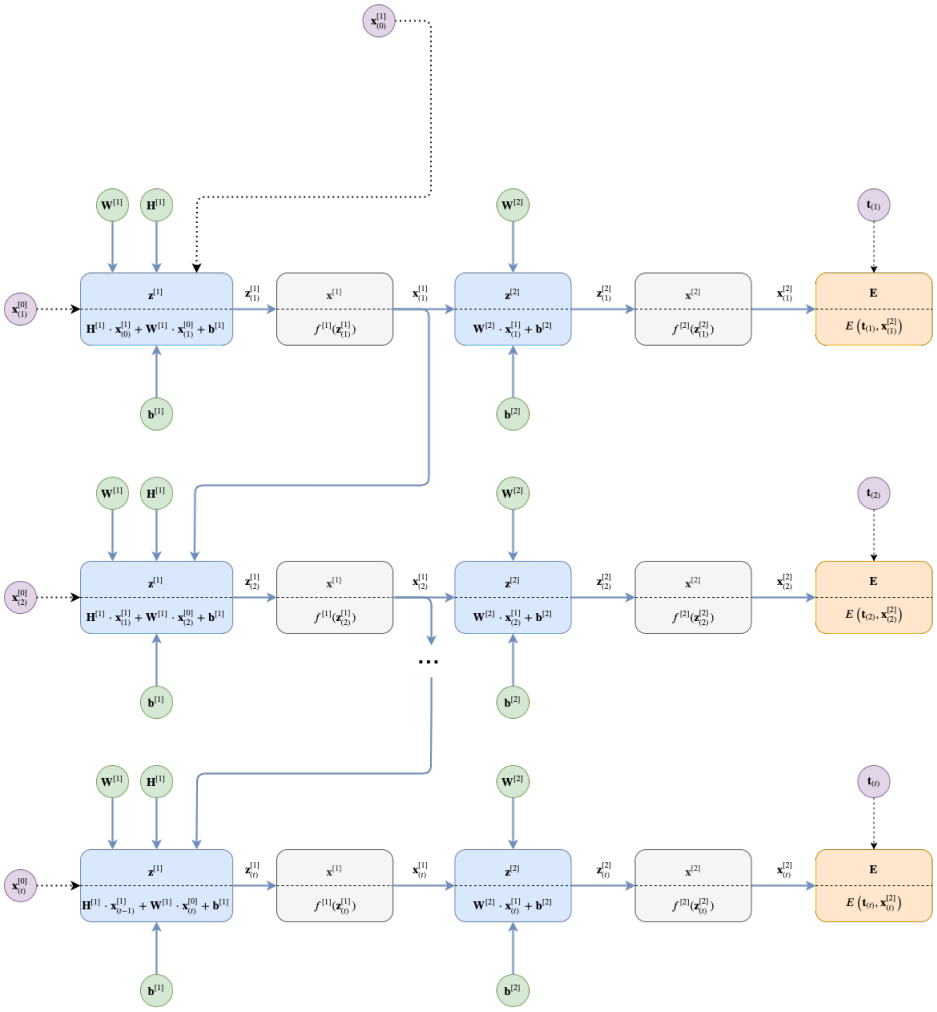
Pedro Balage, Gonçalo Faria, Luis Sa-Couto, Andreas Wichert, André Martins

Introduction

Below we present a simple recurrent network that maps a sequence of inputs to a sequence of targets. In this example, we use only one hidden layer that has a feedback connection.

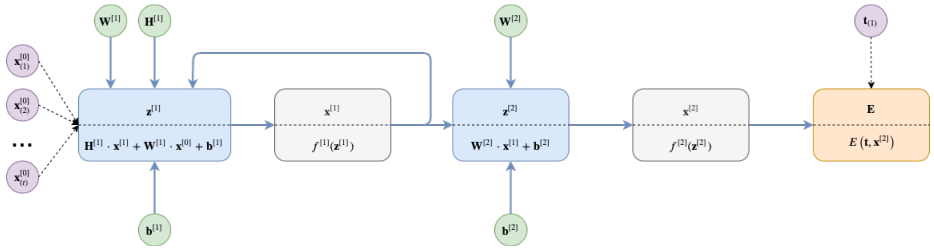


When one thinks about how to apply backpropagation to learn the parameters on such a network it may not seem obvious at first. However, if we unfold the network's function graph, the recurrent network becomes a much like a regular one.

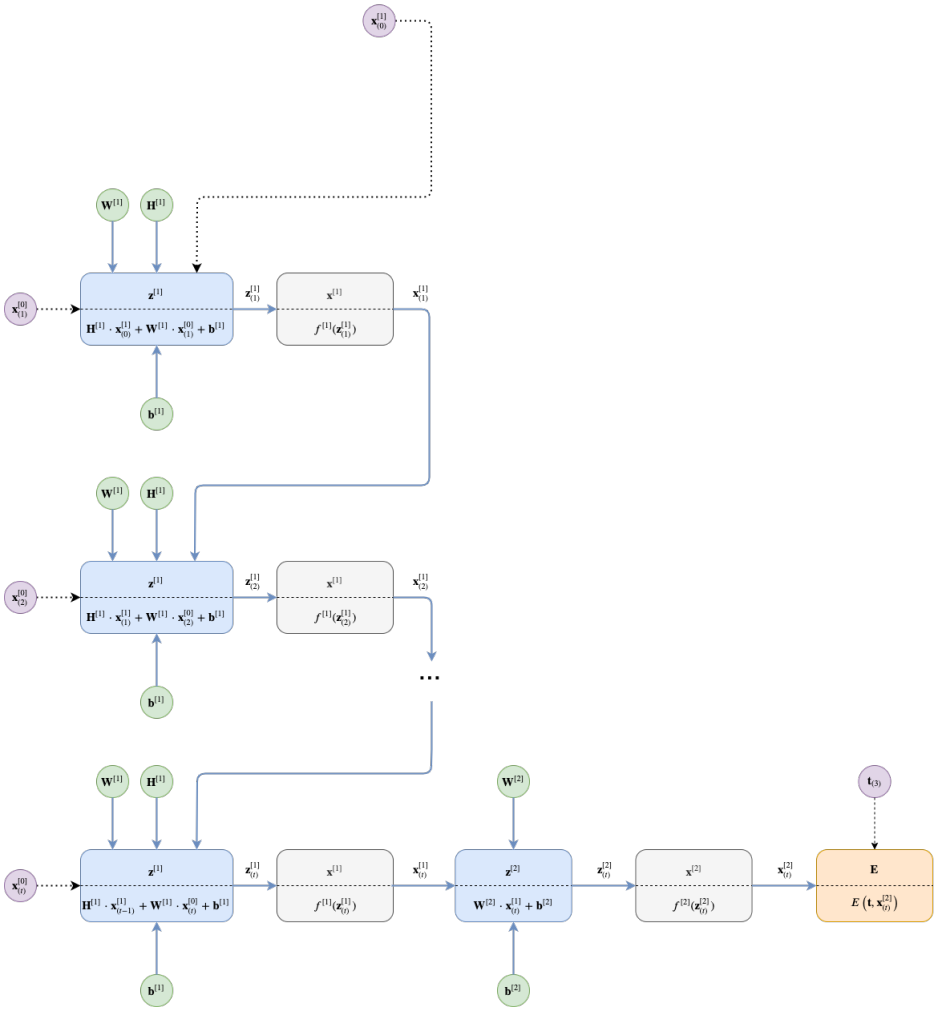


Unfolding corresponds to making copies of the network structure to match the total number of sequence elements. This implies that between copies of the network the weights are shared! This means that $\mathbf{W}^{[1]}$ at the bottom of the unfolded graph is the same as the $\mathbf{W}^{[1]}$ at the top. This weight sharing will influence the way we apply the backward pass. Namely, to update a parameter, we need to sum accross all paths that lead to all copies of that parameter. A requirement of this approach is that we need to initialize $\mathbf{x}_{(0)}^{[1]}$ to feed as hidden state to the first copy of the network.

As an example, we used a many-to-many recurrent network and showed unfolding corresponds to making copies of its structure for every timestep. However, many other architecture templates exist. For example, we could have a many-to-one network. In this case, we will see that sometimes what we have to replicate is just the recursion part. For example, for the network:



the unfolding would be done as follows:



Note that the recursion is copied throughout all timesteps but since there is only one output, we only place an output layer after the last timestep.

Question 1

Consider a recurrent network with one hidden layer to solve a **many-to-many** task. Take into account that:

- The recurrent connections happen between the hidden layers
- The hidden activation function is the hyperbolic tangent
- The output activation function is softmax
- The error function is the cross-entropy between output and target

1. Write down the model equations for forward propagation.

Solution:

Let

- $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_T$ denote the inputs,
- $\boldsymbol{h}_1, \ldots, \boldsymbol{h}_T$ denote the hidden states,

- $\boldsymbol{p}_1, \ldots, \boldsymbol{p}_T$ the label probabilities computed by the model,
- $y_1, \ldots, y_T$ denote the gold outputs, $\boldsymbol{e}_{y_1}, \ldots, \boldsymbol{e}_{y_T}$ their one-hot vector representations,
- $\ell_1, \ldots, \ell_T$ the cross-entropy loss incurred at each time step,
- $\mathbf{W}_{hh}$ the recurrent (hidden-hidden) matrix, $\mathbf{W}_{xh}$ the input-hidden matrix, $\mathbf{W}_{hy}$ the hidden-output matrix, $\boldsymbol{b}_h$ the bias in the hidden layer, and $\boldsymbol{b}_y$ the bias in the output layer.

For all $t \in \{1, \ldots, T\}$, the forward equations are:

$$\begin{aligned}
\boldsymbol{z}_t^{[1]} &= \mathbf{W}_{hh}\boldsymbol{h}_{t-1} + \mathbf{W}_{xh}\boldsymbol{x}_t + \boldsymbol{b}_h \\
\boldsymbol{h}_t^{[1]} &= \tanh(\boldsymbol{z}_t^{[1]}) \\
\boldsymbol{z}_t^{[2]} &= \mathbf{W}_{hy}\boldsymbol{h}_t + \boldsymbol{b}_y \\
\boldsymbol{p}_t &= \text{softmax}(\boldsymbol{z}_t^{[2]}) \\
\ell_t &= \log[\boldsymbol{p}_t]_{\boldsymbol{y}_t} = -\boldsymbol{e}_{\boldsymbol{y}_t} \cdot \log \boldsymbol{p}_t.
\end{aligned}$$

2. Perform a forward propagation for the following sequence of length 2:

$$[\boldsymbol{x}_1, \boldsymbol{x}_2] = \left[\begin{pmatrix} 4 \\ 0 \\ 0 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 8 \\ 2 \\ 0 \end{pmatrix}\right]$$

With targets (gold outputs):

$$[\boldsymbol{e}_{y_1}, \boldsymbol{e}_{y_2}] = \left[\begin{pmatrix} 1 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \end{pmatrix}\right]$$

Initialize all weights and biases to 0.1, using 3 units per hidden layer, initializing the hidden state to all zeros and using $\eta = 1.0$.

Solution:

Initializing the parameters, we have:

$$\mathbf{W}_{xh} = \begin{pmatrix} 0.1 & 0.1 & 0.1 & 0.1 \\ 0.1 & 0.1 & 0.1 & 0.1 \\ 0.1 & 0.1 & 0.1 & 0.1 \end{pmatrix}$$

$$\boldsymbol{b}_h = \begin{pmatrix} 0.1 \\ 0.1 \\ 0.1 \end{pmatrix}$$

$$\mathbf{W}_{hh} = \begin{pmatrix} 0.1 & 0.1 & 0.1 \\ 0.1 & 0.1 & 0.1 \\ 0.1 & 0.1 & 0.1 \end{pmatrix}$$

$$\mathbf{W}_{hy} = \begin{pmatrix} 0.1 & 0.1 & 0.1 \\ 0.1 & 0.1 & 0.1 \end{pmatrix}$$

$$\boldsymbol{b}_y = \begin{pmatrix} 0.1 \\ 0.1 \end{pmatrix}$$

$$\boldsymbol{h}_0 = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}$$

Now, we can use the equations from the previous question to apply forward propagation. Let us start with timestep $t = 1$:

$$\begin{aligned}
\boldsymbol{z}_1^{[1]} &= \begin{pmatrix} 0.1 & 0.1 & 0.1 \\ 0.1 & 0.1 & 0.1 \\ 0.1 & 0.1 & 0.1 \end{pmatrix} \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} + \begin{pmatrix} 0.1 & 0.1 & 0.1 & 0.1 \\ 0.1 & 0.1 & 0.1 & 0.1 \\ 0.1 & 0.1 & 0.1 & 0.1 \end{pmatrix} \begin{pmatrix} 4 \\ 0 \\ 0 \\ 0 \end{pmatrix} + \begin{pmatrix} 0.1 \\ 0.1 \\ 0.1 \end{pmatrix} \\
&= \begin{pmatrix} 0.5 \\ 0.5 \\ 0.5 \end{pmatrix} \\
\boldsymbol{h}_1 &= \tanh \begin{pmatrix} 0.5 \\ 0.5 \\ 0.5 \end{pmatrix} \\
\boldsymbol{z}_1^{[2]} &= \begin{pmatrix} 0.1 & 0.1 & 0.1 \\ 0.1 & 0.1 & 0.1 \end{pmatrix} \tanh \begin{pmatrix} 0.5 \\ 0.5 \\ 0.5 \end{pmatrix} + \begin{pmatrix} 0.1 \\ 0.1 \end{pmatrix} \\
&= \begin{pmatrix} 0.3 \tanh 0.5 + 0.1 \\ 0.3 \tanh 0.5 + 0.1 \end{pmatrix} \\
\boldsymbol{p}_1 &= \operatorname{softmax} \begin{pmatrix} 0.3 \tanh 0.5 + 0.1 \\ 0.3 \tanh 0.5 + 0.1 \end{pmatrix} \\
&= \begin{pmatrix} 0.5 \\ 0.5 \end{pmatrix}
\end{aligned}$$

Now we can apply the same logic to the second timestep.

$$\boldsymbol{z}_2^{[1]} = \begin{pmatrix} 0.1 & 0.1 & 0.1 \\ 0.1 & 0.1 & 0.1 \\ 0.1 & 0.1 & 0.1 \end{pmatrix} \tanh \begin{pmatrix} 0.5 \\ 0.5 \\ 0.5 \end{pmatrix} + \begin{pmatrix} 0.1 & 0.1 & 0.1 & 0.1 \\ 0.1 & 0.1 & 0.1 & 0.1 \\ 0.1 & 0.1 & 0.1 & 0.1 \end{pmatrix} \begin{pmatrix} 0 \\ 8 \\ 2 \\ 0 \end{pmatrix} + \begin{pmatrix} 0.1 \\ 0.1 \\ 0.1 \end{pmatrix}$$

$$= \begin{pmatrix} 1.1 + 0.3 \tanh 0.5 \\ 1.1 + 0.3 \tanh 0.5 \\ 1.1 + 0.3 \tanh 0.5 \end{pmatrix}$$

$$\boldsymbol{h}_2 = \tanh \begin{pmatrix} 1.1 + 0.3 \tanh 0.5 \\ 1.1 + 0.3 \tanh 0.5 \\ 1.1 + 0.3 \tanh 0.5 \end{pmatrix}$$

$$\boldsymbol{z}_2^{[2]} = \begin{pmatrix} 0.1 & 0.1 & 0.1 \\ 0.1 & 0.1 & 0.1 \end{pmatrix} \tanh \begin{pmatrix} 1.1 + 0.3 \tanh 0.5 \\ 1.1 + 0.3 \tanh 0.5 \\ 1.1 + 0.3 \tanh 0.5 \end{pmatrix} + \begin{pmatrix} 0.1 \\ 0.1 \end{pmatrix}$$

$$= \begin{pmatrix} 0.3 \tanh (1.1 + 0.3 \tanh 0.5) + 0.1 \\ 0.3 \tanh (1.1 + 0.3 \tanh 0.5) + 0.1 \end{pmatrix}$$

$$\boldsymbol{p}_2 = \operatorname{softmax} \begin{pmatrix} 0.3 \tanh (1.1 + 0.3 \tanh 0.5) + 0.1 \\ 0.3 \tanh (1.1 + 0.3 \tanh 0.5) + 0.1 \end{pmatrix}$$

$$= \begin{pmatrix} 0.5 \\ 0.5 \end{pmatrix}$$

3. Implement the forward propagation from the previous item in NumPy.

Question 2

One of the capabilities of a RNN is to model language generation. In this exercise, we will use Pytorch to implement a RNN cell and use it to generate random person names.

1. Complete the following code with the RNN cell operations.

```
import torch
import torch.nn as nn

# Vanilla RNN implementation
class RNN(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(RNN, self).__init__()
        # TODO

    def single_forward(self, input, hidden):
        # TODO
        return output, hidden

    def forward(self, input_line_tensor):
        outputs = []
        hidden = self.initial_hidden
        for x_t in input_line_tensor:
            output, hidden = self.single_forward(
                x_t, hidden)
```

```
            outputs.append(output)
        return outputs
```

Solution:

```
class RNN(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(RNN, self).__init__()
        self.hidden_size = hidden_size
        self.i2h = nn.Linear(input_size + hidden_size, hidden_size)
        self.tanh= nn.Tanh()
        self.o2o = nn.Linear(hidden_size, output_size)
        self.initial_hidden = nn.Parameter(
            torch.zeros(1, hidden_size)
        )
        self.softmax = nn.LogSoftmax(dim=1)

    def single_forward(self, input, hidden):
        input_combined = torch.cat((input, hidden), 1)
        hidden = self.tanh(self.i2h(input_combined))
        output = self.o2o(hidden)
        output = self.softmax(output)
        return output, hidden

    def forward(self, input_line_tensor):

        outputs = []

        hidden = self.initial_hidden

        for x_t in input_line_tensor:
            output, hidden = self.single_forward(
                x_t, hidden)
            outputs.append(output)

        return outputs
```

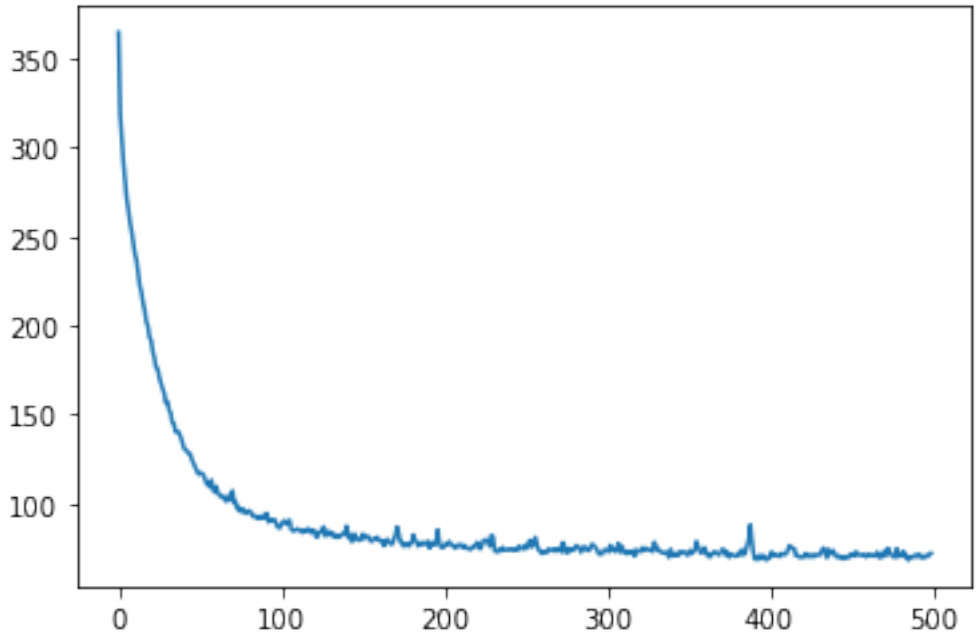
2. Explore how to to load the data from the file `names.txt` as training examples. You may use the following functions to help you.

```
# all letters to work with the Portuguese names from the train file
all_letters = 'abcdefghijklmnopqrstuvxwyzáàãçéêíóõôú'
n_letters = len(all_letters) + 1 # Plus EOS marker

# One-hot matrix of first to last letters (not including EOS) for input
def inputTensor(line):
    tensor = torch.zeros(len(line), 1, n_letters)
    for li in range(len(line)):
        letter = line[li]
        tensor[li][0][all_letters.find(letter)] = 1
```

```
    return tensor

# LongTensor of second letter to end (EOS) for target
def targetTensor(line):
    letter_indexes = [all_letters.find(line[li]) for li in range(1, len(line))]
    letter_indexes.append(n_letters - 1) # EOS
    return torch.LongTensor(letter_indexes)
```

3. Using the previous functions, train a model for 1,000 epochs using negative log-likelihood, a hidden size of 128 and a learning rate of 0.0005. Assume all necessary initialization with 0. Report the training loss curve.

   Solution:



4. Explore how to generate names starting from a random letter. Use the functions topk to guess the next letter in each iteration. Did you get good results? Discuss the results you got.

   Solution:

   With the actual implementation there is a high change that the model "memorize" the names from the trainset by reinforcing the sequence of letters seen in the data. In order to avoid this problem we may want to explore the use of regularizers and also the selection of the second or third best letter in the sequence decoding (`topk(2)`, `topk(3)`).

Question 3

Now it's time to try the Recurrent Neural Networks on real data and compare it with other approaches previously seen in our practical sessions.

Let's use PyTorch and RNNs to perform sentiment analysis with the IMDB database.

1. Using torchtext, load the IMDB dataset and explore the data iterator with the following code.

```
import torch
from torchtext.legacy import data , datasets

TEXT = data.Field()
LABEL = data.LabelField(dtype = torch.float)

train_data, test_data = datasets.IMDB.splits(TEXT, LABEL)

train_data, valid_data = train_data.split()

# build the vocabulary
TEXT.build_vocab(train_data, max_size = 20_000)
LABEL.build_vocab(train_data)


BATCH_SIZE = 64
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

train_iterator, valid_iterator, test_iterator = data.BucketIterator.splits(
    (train_data, valid_data, test_data),
    batch_size = BATCH_SIZE,
    sort_within_batch = True,
    device = device)
```

2. Using the RNN class below, train a sentiment classifier for 5 epochs using an embedding layer with dimension 100 and hidden layer with dimension 256. Use the `Adam` optimizer and the `BCEWithLogitsLoss`. Report the final accuracy.

```
import torch.nn as nn

class RNN(nn.Module):
    def __init__(self, vocab_size, embedding_dim, hidden_dim, output_dim):

        super().__init__()
        self.embedding = nn.Embedding(vocab_size, embedding_dim)
        self.rnn = nn.LSTM(embedding_dim, hidden_dim)
        self.fc = nn.Linear(hidden_dim, output_dim)

    def forward(self, text):

        #text = [sent len, batch size]
        #embedded = [sent len, batch size, emb dim]
        embedded = self.embedding(text)

        #output = [sent len, batch size, hid dim * num directions]
        #hidden = [num layers * num directions, batch size, hid dim]
        #cell = [num layers * num directions, batch size, hid dim]
        output, (hidden, cell) = self.rnn(embedded)
```

```
        #hidden = [batch size, hid dim * num directions]
        hidden = hidden[-1,:,:]

        return self.fc(hidden)
```

3. Change the network architecture to use a bidirectional LSTM and report the new results. Discuss how would you improve the results even more.

   Suggestion: You may want to run different experiments in Google Colab to have a faster execution using a GPU environment.

   Solution: Using a bidirectional LSTM require some changes in the code as the output from both directions need to be concatenated.

   In order to improve the results, other strategies may be considered:

   - Start with pre-trained embeddings
   - Use a padded sequence
   - Add a dropout
   - Change the hyperparameters (number of layers; layer sizes, etc)